2008

Verifying sensor network security protocol implementations

Youssef Wasfy Hanna
Iowa State University

Verifying sensor network security protocol implementations

by

Youssef Hanna

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Hridesh Rajan, Major Professor
Samik Basu
Wensheng Zhang

Iowa State University

Ames, Iowa

2008

DEDICATION

To my parents and Mina.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. I would like to thank Hridesh Rajan for his guidance, patience and support throughout this research and the writing of this thesis. Thanks are due to the National Science Foundation for financially supporting this project.

I would like to thank my committee members Wensheng Zhang and Samik Basu for their efforts and contributions to this work. Additionally, I would like to thank the reviewers of Wisec 2008 conference for their insightful feedback. Discussions with the Doctoral Symposium committee members of the ESEC/FSE 2007 conference were of great help to me. I would like to specially thank Richard Kemmerer for his great advices. I would like to extend my thanks to all the members of Laboratory of Software Design for offering constructive criticism and timely suggestions during research.

I am very grateful to my parents and my brother Mina for their moral support and encouragement throughout the duration of my studies. I am grateful to my friend Dana Awwad for her support during writing my journal submission.

CHAPTER 1. Introduction

Sensor networks are networks of small devices used for sensing and recording information about environmental changes. Applications of sensor networks include civilian applications (i.e. fire detection in forests [54], monitoring volcano eruptions [52], etc) as well military applications such as battlefield surveillance [2]. Due to the fact that sensor networks operate unattended for long periods of time, security protocols are required to protect the information broadcast through the network. As the sensor nodes are resource constrained, new security protocols have been specially designed to protect those networks.

Establishing the correctness of the security protocols has been a daunting task. Current techniques for verification are classified under two categories: simulation and formal methods. While simulation can detect some errors in the security protocol implementations, simulators do not cover all possible behaviors of the implementation, thus some subtle errors might not be discovered in the simulation process. Formal methods such as model checking [15] have solved this problem by verifying that a property is satisfied against an abstract specification (model) of the security protocol, thus covering all possible behaviors of the model.

There are four problems with model checking that make it hard to apply in the verification of sensor network security protocols. Firstly, building models is a time consuming task that may take more time than implementing the system itself [22]. Secondly, building the models required for the verification requires expertise in modeling languages, which may prevent domain experts from attempting such tasks [45]. Thirdly, the abstraction required to build the model may abstract some of the details of the implementation that may contain security flaws, which in turns will go undetected [5]. Finally, there is no guarantee that the abstract specification of the protocol is actually implementing the intended behavior of the protocol. So, even though

verifying the model can show that the protocol is secure, the implementation of the protocol may still contain some flaws [5].

This work addresses these problems. In this thesis, I present our framework *Slede* for automatic formal verification of sensor network security protocol implementations written in nesC [18] [1]. *Slede* does not require upfront model construction thereby significantly easing the task of sensor network developer. Instead, a skeleton model is automatically extracted from the *nesC* implementation of the security protocol. This extracted skeleton model is then automatically composed with intrusion models and desired network topologies to create a complete verifiable model. The key technical contributions of our work are:

- An automated technique for extracting a skeleton model of the protocol from its nesC implementations,
- a light-weight language design to semi-formally describe the protocol specification,
- an approach for customizing the skeleton model of the protocol with the help of the protocol specification and inbuilt intrusion detection models to generate a complete verifiable models, and,
- a technique for mapping the results of verification back to the domain terms.

The main challenge in this work was to overcome the state space explosion problem. Our annotation language deals with this problem by providing necessary information about the protocol specification to the intruder model generator, which uses this information to generate a smart intruder, thus decreasing the state space.

We validated the framework *Slede* by using it to verify implementations of two sensor network security protocols. The framework was able to confirm known flaws in these two protocols.

Our approach is sound and complete within bounds. In other words, if the framework returns a fault scenario in the protocol for a certain network topology and a given size, then

[1] These ideas were first proposed and discussed in our publication [23]. This thesis is a revised and an extended version of this paper.

there is indeed a fault in such a network. Otherwise, no faults in the protocol are present for this network topology and given size or less (if the verifier halts).

1.1 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 1: The sensor networks are often deployed unattended to monitor and record environmental changes. Security protocols have been designed to protect these networks. Different verification methods ranging from testing, simulation and formal methods have been used to ensure that security protocols are satisfying their goals. This chapter presents a brief overview of the problem of verifying security protocols.

Chapter 2: This chapter gives an introduction about sensor networks, their applications and the need to secure them. Model checking as a verification technique is also described in this chapter.

Chapter 3: Discussion about existing approaches for verifying security protcols is presented in this chapter. Different works on verification from abstract specification as well as verifying real code through model extraction are also discussed.

Chapter 4: The detailed problem definition for which this thesis proposes a solution as well as motivation for verifying the sensor network security protocol implementation as opposed to verify the specification are described in this chapter.

Chapter 5: Our proposed framework *Slede* for automating the verification of the sensor network security protocol implementations is presented in this chapter. With the help of information about the protocol such as message structure and topology presented using our annotation language, *Slede* extracts a verifiable Promela model from the implementation. This chapter describes the proposed framework and the annotation language in details.

Chapter 6: This chapter describes the results of verifying the sensor network security protocols directly from the implementation using our proposed framework. The implementations of two sensor network security protocols were verified using our framework. Verification results from the extracted models demonstrate the effectiveness of our proposed work.

Chapter 7: This chapter concludes the thesis. The problems of verifying security protocols from the abstract specification and the benefits of verifying the implementation of the protocols automatically using our proposed framework are discussed here. The chapter also presents future work such as extending the framework to include verification of the network lifetime, introduction of the node compromise model and optimization techniques to reduce the state space.

CHAPTER 2. Background

A brief discussion on sensor networks and model checking is presented in this chapter.

2.1 Sensor Networks

A sensor network is a collection of small size, low power and low cost sensor node that have limited storage, computation and communication capabilities. The nodes are often operated unattended to monitor and record information about environmental changes. The applications of sensor networks fall under two categories: civilian and military. Examples of civilian applications include fire detection in forests [54], habitat monitoring [33], etc. Military applications include battlefield surveillance, battle damage assessment, targeting, etc [2].

The operating environment of sensor networks is often hostile, thus security protocols are required to ensure secure communication between sensor nodes. Due to the low power, communication and storage capacity of the sensor nodes, securing the sensor network is challenging. With sensor nodes of battery power as low as 1 joule per cubic mm [51], current cryptographic protocols are often unusable. Therefore, new security protocol have been implemented to fit the sensor nodes capabilities (see [8] for a survey).

2.2 Model Checking

In model checking, the system to be verified is represented as a finite-state model M, a model that represents all possible scenarios of the system behavior. Given a property φ, the verifier checks all the paths of the model M for any branch that violates the property. If no branch in the model violates this property then the system is said to satisfy the property

(denoted as $M \models \varphi$). Otherwise, the verifier outputs the branch that has violated the property, known as the *counterexample*.

The main challenge in applying the model checking verification technique is the state explosion problem [50]. When the number of states in the model increases beyond a certain limit, the verifier is unable to go through all of the branches of the model and thus may never halt. Many approaches have been designed to reduce the state space of the model (i.e. partial order reduction [3, 19, 21], bit-state hashing [24]), yet the state explosion problem remains the greatest challenge when applying model checking technique.

CHAPTER 3. Related Work

In this chapter we discuss previous and current work in the field of specification and verification of security protocols.

3.1 Specification and Verification Techniques

Security protocols specification methods range from informal narrations of the messages exchanges to formal assertions of protocol properties [1]. The problem with describing the security protocol using natural language is that though it is easy, it lacks rigor. The same protocol could be easily interpreted in different ways when read by different people. The most well known and influential approach for formally specifying and verifying security protocols is the BAN logic, developed by Burrows, Abadi and Needham [10]. The key idea is to reason about the state of beliefs among principals in a system.

A number of language-base approaches for specification of security protocols have been developed. The common authentication protocol specification language (CAPSL) [38] is one of the most visible work in this area. The problem CAPSL was to solve is that for every verification mechanism for security protocols, the protocol has to be specified in terms of the specification language required for this mechanism. The idea behind CAPSL was to create a unified specification language where all different specification languages are translated to it, thus working as an intermediate language.

There is a significant body of research on verifying security protocols. Meadows developed the NRL protocol analyzer for the analysis of cryptographic protocols [36]. The NRL protocol analyzer was used to find flaws in a number of cryptographic protocols including selective broadcast protocol by Simmons [47], Resource Sharing Protocol by Burns and Mitchell [9],

re-authentication protocol by Neuman and Stubblebine [40], etc. Longley and Rigby also developed a tool and demonstrated a flaw in a banking security protocol [32]. Kemmerer [26] used general-purpose formal methods technique as tools to verify cryptographic protocols. For a detailed summary of verification techniques, please refer to a survey by Rubin and Honeyman [46], Meadows [35], Gritzalis et al. [48], and a more recent survey by Buttyan [11].

3.2 Verification of Security Protocol Implementations

The closest work related to our approach is by Bhargavan *et al.* [5] and by Goubault-Larrecq and Parrennes [22]. Bhargavan *et al.* [5] present an approach for verifying protocol implementations written in F# using ProVerif [6], a theorem prover as the underlying mechanism. Our work is different in two dimensions: first, we are verifying protocol implementations written in nesC, and second, we use a model checker as the underlying technology.

Goubault-Larrecq and Parrennes [22] present an approach for verifying protocol implementations in C. Their approach models secrecy properties as reachability properties of the C implementation and analyzes these properties. A simple pointer analysis technique is used to keep the verified model as close as possible to the actual implementation. Unlike our approach that provides support for the entire nesC language, this approach is useful only for C implementations; however, the insights described by Goubault-Larrecq and Parrennes [22] could be used to enhance the underlying verification technique for our framework.

Kothari *et al* [27] present their tool *FSMGen* for extracting finite state machines from TinyOS programs. They apply symbolic execution in order to deal with the event-driven nature of TinyOS, and then apply predicate abstraction to reduce the size of the generated model; however, the goal of their work is different from ours. The main goal of their approach is to derive the state machine from TinyOS applications to infer user-readable high-level representation of the TinyOS program that can be used for automatic program verification (race conditions, etc). Our framework, in addition to the model extraction, is able to generate intruders, merge them with the extracted model and verify the resulting model for security breaches.

Tobarra *et al.* [49] propose an approach for verification of sensor network security protocols using model checking. In their approach, they verify the models of the protocols written in HLPSL [12] modeling language using the model checking tool Avispa [4]. They were able to discover attacks in two security protocols; however, unlike *Slede*, they verify the protocols against models written manually, which does not provide any guarantee that the implementation of the protocol is correct.

CHAPTER 4. Problem Definition

Establishing the correctness of security protocol implementations continues to be a daunting task as their complexity continue to increase. In the past, even widely-studied security protocols are shown to have faults that are detected much later [10, 47, 36].

Verifying sensor network security protocol implementations is even harder. The reason behind this is that these implementations are developed for a severely resource constrained environment. Efficiency and code size are more likely to weigh over readability and understandability, which in turn increases the likelihood of inconsistencies and errors.

The demand for robust performance in unattended deployment scenarios in hostile environments makes this problem more severe, which necessitates uncovering any errors as early in the development process as possible because on-site bug-fixes and updates are often impossible or, at the very least, prohibitively costly. Therefore, detecting and removing errors from sensor network security protocol implementations is extremely important.

4.1 Verification using Testing and Simulation

The variety of methodologies that have been suggested for verification may be classified under two broad categories, simulation and formal methods. Functional simulation of the implementation using simulators such as TOSSIM [29] and/or test runs of the protocol implementation on sensor network test beds are the primary techniques used within the research community to verify implementations due to its simplicity and scalability. However, exhaustive simulation is often impractical, and the likelihood that these tests will uncover subtle errors is diminishing.

For instance, let our goal be to detect routing loops in an implementation of the AODV (Ad-

hoc On Demand Distance Vector) protocol [42], a loop-free routing protocol for ad-hoc networks that guarantees that the network is always free of loops. As the protocol has no mechanism to detect or recover from loops, testing the implementation (as well as the specification) of the protocol is necessary, otherwise the protocol will fail [16]. While testing may detect some routing loops, there is no guarantee that the test cases have detected all routing loops. An incomplete test case can easily miss one possible execution of the protocol where the routing loop lies. Similarly, simulating such protocol implementation cannot ensure that all routing loops have been discovered, because the simulator does not necessarily traverse all possible executions of the protocol.

To avoid such problems of having some behaviors uncovered by testing or simulation, formal methods such as model checking have emerged as verification techniques that promise to verify against all possible behaviors.

4.2 Model Checking vs Simulation/Testing

Verification from abstract specifications using techniques such as model checking has been used to find flaws in cryptographic protocols [34] (see [37] for a survey). The main benefit of model checking over verification techniques like simulating using TOSSIM and manual inspection is that the model verified by the model checker covers all possible scenarios of the behavior of the system. Unlike simulation that does not provide exhaustive coverage, verification using model checking gives a more thorough analysis of the system by checking satisfiability of the requirement through every branch of the model representing the system. For instance, in the example of verifying AODV protocol for routing loops, since the model checker will go through all possible executions of the protocol, the model checker should be able to detect all routing loops if there are any, as opposed to simulation and testing where they may not discover them due to the possibility that some possible executions of the protocol might go unverified.

4.3 Problems in Applying Model Checking

Applying model checking technique for verification, however, requires non-trivial efforts primarily because model checking tools often require a specification (model) of the system under verification. This specification is written in a specialized language. Learning this language itself can be a daunting task. This task is further complicated by the impedance mismatch between the implementation language and the modeling language. For example, while the dominant implementation language for sensor network applications (*nesC*) uses an event-based paradigm, an example modeling language (*Promela*) uses message-driven paradigm. Moreover, as described in Chapter 1, constructing a model from an implementation of the protocol may not be desired because of the time consumed to build the models, the effort required to learn the modeling language, the possible discrepancy of behavior between the model and implementation as well as the flaws that might go undetected due to the abstraction required to build the model.

To illustrate the problem with building the model required for applying model checking, we describe in this section the verification of an example sensor network security protocol, the polynomial pool based pairwise key establishment protocol [31].

The polynomial pool based pairwise key establishment protocol includes two phases: system initialization before network deployment and pairwise key establishment after deployment. Before deployment, the network controller picks n symmetric bivariate polynomials $f_i(x, y)$ $(i = 0, \cdots, n-1)$; every sensor node with ID A is preloaded with $m < n$ univariate polynomials $f_{i_k}(A, y)$ $(k = 0, \cdots, m-1)$, which are shares of m out of n aforementioned bivariate polynomials.

After deployment, if neighboring nodes A and B have shares derived from the same bivariate polynomial, for example, $f_0(A, y)$ and $f_0(B, y)$, they can directly establish $f_i(A, B) = f_i(B, A)$ as their pairwise key. Otherwise, A and B will find one or more helping nodes I_1, I_2, $\cdots$, I_s such that, each pair of adjacent nodes on the chain A, I_1, I_2, $\cdots$, I_s have shares derived from the same bivariate polynomial, and thus can set up a pairwise key. Then, A picks a new key, encrypts it with the pairwise key shared with I_1, and sends it to I_1. I_1 and the following nodes

```
1 event uint8_t* Channel.receive(uint8_t *msg){
2  uint16_t node_s,node_d;
3  uint16_t key[4],keys[4],keyd[4];
4  uint8_t cks[2],ckd[2];
5  uint8_t i,j,type;
6  bool same,sames,samed,cp;
7  ...
8  // every sensor that received the msg
9  // checks itself
10 if((type==4)&&(node_s!=sID)&&(node_d!=sID)) {
11  msg[5+ss+ss] = msg[5+ss+ss]-1;
12  sames = check(msg+5+ss,secret,cks);
13  samed = check(msg+5,secret,ckd);

15  if((sames==1)&&(samed==1)){
16   // node_my send *cks *shares to node_s
17   j=cks[1];
18   i=ckd[1];
19   call ComputeKey.compute(
20    (uint8_t *)&secret[j],node_s,(uint8_t *)keys);
21   call ComputeKey.compute(
22    (uint8_t *)&secret[i],node_d,(uint8_t *)keyd);
23   // generate a random number
24   key[0] = call Random.rand();
25   key[1] = call Random.rand();
26   key[2] = call Random.rand();
27   key[3] = call Random.rand();
28   // encrypt key[0] with keys and keyd separately
29   call Primitive.encrypt(
30    (uint8_t *)keys,(uint8_t *)key,(uint8_t *)keys);
31   call Primitive.encrypt(
32    (uint8_t *)keyd,(uint8_t *)key,(uint8_t *)keyd);
33   msg[0] = 5; // type5: send path key
34   memcpy(msg+1,(uint8_t *)&sID,2); // src node
35   memcpy(msg+3,(uint8_t *)&node_s,2); // dest node
36   msg[5] = ckd[0];
37   memcpy(msg+6,(uint8_t *)keyd,8);
38   msg[14] = cks[0];
39   memcpy(msg+15,(uint8_t *)keys,8);
40   if(sendflag==0) {
41    call Channel.send(node_s,msg);
42    sendflag = 1;
43   }
44   ...
45  }
46 }
47 ...
48 }
```

Figure 4.1 A snippet of the polynomial pool based pairwise key establishment protocol implementation [31]

in the chain uses the same approach to secretly transmit the new key hop-by-hop towards B. This way, a pairwise key can be established between A and B.

A part of the nesC implementation of the polynomial pool based pairwise key establishment protocol is shown in Figure 4.1. This part is responsible for the path discovery. First, we can see that the implementation is much more complicated compared to the abstract description of the protocol. Even though the manual inspection and mathematical analysis of the abstract specification of the protocol might ensure security of the protocol, the implementation of the protocol may not accurately implement the abstract protocol, thus leaving room for subtle errors that can be exploited by adversaries. Moreover, verification of the implementation using simulators like TOSSIM [29] or using testbeds may not exhaustively test all paths in the protocol implementation, thus leaving room in the untested paths for possible attacks.

A model for this protocol in the Promela language is shown in Figure 4.2. This model can be verified by the model checker Spin [25]. One challenge in building such model is to emulate the physical characteristics of sensor networks in the model that plays a key role in verification. For instance, to emulate the wireless channels, the receiver of the message should not prevent other sensors from receiving it as well.

If Promela's inbuilt construct for sending and receiving message is used to emulate sending

```
1 do
2   /*Receiving a msg*/
3 :: channel.containsMsg==1 ->
4    atomic {
5      msg.src = channel.msg.src;
6      msg.dest = channel.msg.dest;
7      msg.info = channel.msg.info;
8      ...
9    }
10   if
11   :: msg.type == 4
12       && msg.src == agent_id
13       && msg.dest == agent_id ->
14       ...
15       /* Non-determinism for
16          emulating random call*/
17       if
18       :: key[0] = 1;
19       :: key[0] = 2;
20       :: key[0] = 3;
21       ...
22       fi;
23       /* same for key[1], key[2]... */
24       ...
25       /* Sending message */
26       atomic{
27          channel.msg.src = msg.src;
28          ...
29       }
30     :: else -> ...
31     fi;
32 od;
```

Figure 4.2 Promela model for the code in Figure 4.1

and receiving message in this protocol, the message will be removed from the channel when a receiver receives the message. Thus, a modified version of receiving like the one used in Figure 4.2 (lines 3-9) is needed. Furthermore, some mechanism to emulate collision and mutual exclusion is also needed. Here the `atomic` construct of Promela is used to ensure these properties. This construct ensures that all statements between line 5 and 8 are executed as an atomic transaction.

4.4 Model Extracting from Implementation

The moral of the story is that, even though model checking allows us to conduct more rigorous verification compared to simulation and test-beds, a simple error in building the model can easily deviate the behavior of the model from the intended behavior.

An approach that provides technique for automatically extracting a model from protocol implementation and verifies them using existing model checkers seems to solve these problems. Once the automatic extraction approach is verified to be correct, the models extracted by it are guaranteed to faithfully represent the implementation. Thus there are no inconsistency issues between the model and the protocol implementation during the model construction.

Such models can be kept synchronized with the actual implementation by a simple regeneration of the model after the implementation has changed. Such regeneration could be as simple as compiling the protocol implementation and can also be incorporated into the compilers.

Furthermore, such approaches make the benefits of rigorous formal verification technique

available to a developer without requiring them to learn the intricacies of formal techniques and their specification languages, which in turn significantly reduces the overhead of protocol implementation verification.

Our approach is an example of such technique. In the next chapter, we describe our framework for verifying nesC implementations of sensor network security protocols.

CHAPTER 5. Approach

Our framework provides automatic verification of sensor network security protocols by extracting models from the protocol implementation. In addition, the framework automatically generates intruder models that are necessary for verification of the security protocols. It is built on top of the nesC compiler version 1.1.1 [39] and uses the Spin model checker [25] as the backend.

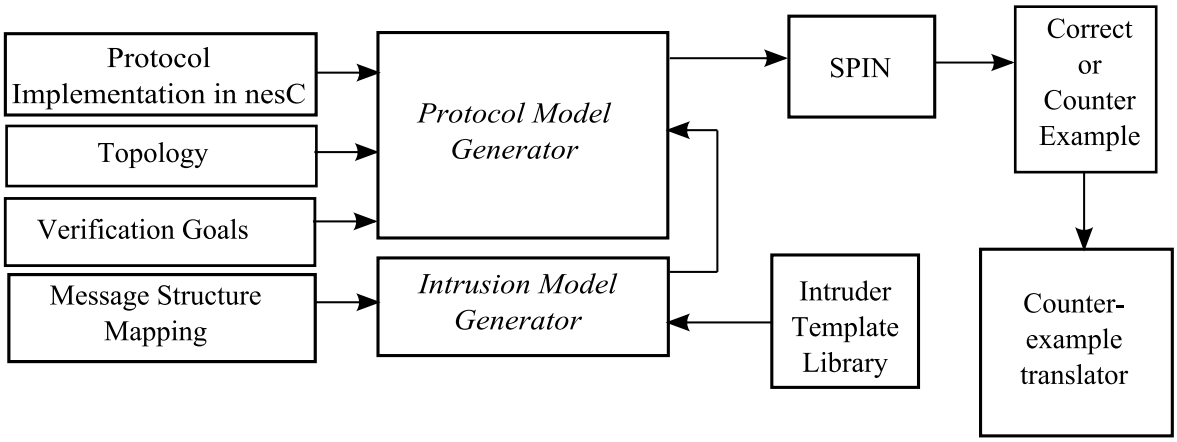

Figure 5.1 Overview of *Slede*

The overview of *Slede* is shown in Figure 5.1. The framework takes the source code of the protocol as input. In order to verify a protocol implementation, besides the source code, the framework requires a small amount of extra information such as the structure of messages used in the protocol, a deployment topology, and properties that need to be verified about the protocol. This information is provided in a different file with extension `sld`. The generated protocol model is merged with the generated intruder model and then verified using the

Spin model checker [25]. If there is a violation of the protocol objective, the counterexample generated by Spin is translated into nesC statements using the counterexample translator.

In this chapter, we describe the annotation language required for protocol verification from implementation as well as the main components *Slede*: the Protocol Model Generator, the Intruder Model Generator and the Counterexample Translator.

5.1 Annotations in Slede

Verification of security protocols requires the presence of a principal that acts maliciously to ensure that the protocol satisfies its goals with the presence of malicious behavior. In order to automatically generate an intruder that behaves maliciously, the framework requires some information about the protocol. Please note that the framework does not infer the protocols from the nesC code (i.e. the framework does not infer the message sequencing of the protocol from the implementation). What the framework does is verify that the implementations of the security protocols satisfy their goals in the presence of malicious behavior.

In order to convey such information to *Slede* to generate the malicious behavior, we developed an annotation language for describing such information required by our framework. This language is also used to describe the security goals against which the protocol should be verified as well as some information required to bound the verification process such as the network topology and the number of iterations of the protocol.

In this section, we describe the annotation language for *Slede*. The description includes syntax, a small example, and informal semantics of the constructs.

5.1.1 Abstract Syntax

The abstract syntax of our annotation language is shown in Figure 5.2. A verification configuration (⟨V⟩) consists of a sequence of message declarations (⟨mdecl⟩*), followed by the message type mapping (⟨tmap⟩), followed by the bound on the number of iterations of protocol execution (⟨iter⟩), followed by a sequence of node definitions (⟨ndef⟩) that specifies the topology, followed by the set of objectives (⟨obj⟩). The objective is the property that is to

⟨V⟩ ::= ⟨mdecl⟩* ⟨tmap⟩ ⟨iter⟩ ⟨ndef⟩ ⟨obj⟩;
⟨mdecl⟩ ::= `message` ⟨mtype⟩ `mapsto` ⟨t⟩ { ⟨term⟩* }
⟨term⟩ ::= `private`* ⟨tname⟩ `mapsto` ⟨f⟩ ;
⟨tmap⟩ ::= `msgtypes mapsto` ⟨t⟩.⟨f⟩ { ⟨mmap⟩* }
⟨mmap⟩ ::= ⟨d⟩:⟨mtype⟩;
⟨iter⟩ ::= `iteration:`⟨d⟩;
⟨ndef⟩ ::= `node` ⟨n⟩ { ⟨rdef⟩* }
⟨rdef⟩ ::= ⟨n⟩;
⟨obj⟩ ::= `objective` ⟨oname⟩ { (⟨o⟩ ;)* }
⟨o⟩ ::= ⟨condition⟩ | `!` ⟨o⟩ | `(`⟨o⟩`)` | ⟨o⟩ `&&` ⟨o⟩ | ⟨o⟩ '`||`' ⟨o⟩ | ⟨o⟩`->`⟨o⟩
⟨condition⟩ ::= ⟨intruder⟩* ⟨mname⟩.⟨iname⟩.⟨fname⟩`(`⟨form⟩*`)`
⟨intruder⟩ ::= `Intruder:`
⟨form⟩ ::= ⟨mtype⟩ | ⟨d⟩

where

m, n	$\in$	$\mathcal{N}$, the set of node names
$mtype$	$\in$	$\mathcal{MT}$, the set of message types
$tname$	$\in$	$\{sender, receiver, data\}$
e, f	$\in$	$\mathcal{F}$, the set of field names in implementation
c, d	$\in$	$\mathcal{I}$, the set of integers
$oname$	$\in$	$\mathcal{O}$, the set of objective names
$mname$	$\in$	$\mathcal{MN}$, the set of module names in implementation
$iname$	$\in$	$\mathcal{IN}$, the set of interface names in implementation
$fname$	$\in$	$\mathcal{FN}$, the set of function (command/event) names in implementation

Figure 5.2 Abstract syntax for the core annotation language

```
1  message Data mapsto IntMsg{
2  sender mapsto src;
3  receiver mapsto dest;
4  data mapsto info;
5  private data mapsto authen;
6  }
7  message Ack mapsto IntMsg{
8  sender mapsto src;
9  receiver mapsto dest;
10 }
11 msgtypes mapsto IntMsg.type {
12 1: Data;
13 2: Ack;
14 }
15 node 0 { 1; }
16 node 1 { 0; }
17 objective Auth{
18 Intruder:SensorM.Send.send(Data)->
19   !SensorM.Send.send(Ack)
20 }
```

Figure 5.3 An Example Verification Configuration

be verified about the protocol with respect to the specified configuration. We explain message declarations, node definitions, and other pieces of the syntax in the following subsections.

In the concrete syntax, the verification configuration is defined in the file of extension `sld` as shown in Figure 5.3. The special words such as `message` are not keywords; they only have special meanings in this context.

5.1.2 Message declarations

A verification configuration in *Slede*'s annotation language may contain a sequence of message declarations that represent messages exchanged between nodes. A message declaration has exactly one message type (⟨mtype⟩) named in the header `message`, followed by the keyword `mapsto`, followed by ⟨t⟩, which is a structure defined in the implementation that is used for exchanging messages between principals. The `mapsto` clause establishes a correspondence between the protocol specification and implementation.

A message declaration may contain a sequence of terms (⟨term⟩*). A ⟨term⟩ is a term

name, which can be one of the special words $\{sender, receiver, data\}$, followed by the keyword `mapsto`, followed by a field ⟨f⟩, which is a field in the structure defined in the implementation that is used for exchanging messages between principals. The types of fields that are mapped to each other should be the same. If the term mapping starts with the word `private`, then this field is the Message Authentication Code (MAC) used for authenticating the message. In this case, the intruder checks first if it has the authentication key for this message. If it has it, then it modifies on the message and sends it; otherwise it just saves it. If the term mapping does not start with `private`, then the intruder can simply read the content of the field.

An example message declaration is given in Figure 5.3 (lines 7-10), where a message type `Ack` is defined. This message type in specification is mapped to the structure `IntMsg` in the implementation. The example defines that a message of this type will contain two integer fields representing the sender and the receiver of the acknowledgement. These fields are mapped to fields `src` and `dest` of the structure `IntMsg`.

The message declaration `Data` in Figure 5.3 (lines 1-6) is an example of a message including a MAC for which the intruder needs a key (field `authen` in line 5). Similar to `Ack`, this message is also mapped to the structure `IntMsg` in implementation. This example shows that two different message types may be implemented using the same structure in the implementation. We show in Section 5.1.3 how to distinguish different message types that are mapped to the same message structure in the implementation. The way to read this message declaration is "`Data` message contains `sender` address, `receiver` address and authenticated `data`".

5.1.3 Differentiating between Message Types

One common approach in implementing protocols in nesC is that only one message structure is used for all different types of messages used in the protocol. To differentiate between different types of messages, usually one field of the message structure in the implementation is used to identify the type of the received message. In the annotation language, this field is referenced using the `msgtypes` special word, followed by `mapsto`, followed by ⟨t⟩, which is a structure defined in the implementation that is used for exchanging messages between princi-

pals, followed by the field `f` responsible for identifying the type of messages, followed by the mapping (⟨mmap⟩) that maps values of the field ⟨f⟩ to the message types ⟨mtype⟩s.

In Figure 5.3, we can see that both message declarations `Data` and `Ack` are mapped to the same structure `IntMsg`. The field `type` of the structure `IntMsg` (lines 11-14) is the field responsible for identifying if the type of the message is either `Data` or `Ack` (where the value of `type` will be either 1 or 2 respectively).

5.1.4 Topology

The major challenge in extracting a model from the implementation is to generate a model of small number of states as possible. If the number of states (also known as state space) increases beyond a certain limit, the model checker will not be able to verify all the model and may never halt. This problem is known as *the state explosion problem*, which is a major challenge when applying the model checking technique. The number of states is directly proportional to the possible executions of the protocol.

To avoid the problem of state explosion, our current prototype allows for verifying the protocol with one topology at a time. This way, we are reducing the execution possibilities (as opposed to checking that every broadcast message was received by all sensors if verification is done against all possible topologies), which reduces the number of generated states. The annotation language allows the user to define the topology of the network against which the protocol will be verified.

To define the topology, every node `n` is declared using the special word `node`, followed by `rdef`, which defines the reachable nodes to `n`.

In lines 15-16 of Figure 5.3, we define the nodes to be involved in the protocol using the special word `node`. After the node name, the nodes to which the node is connected are stated. In this example, we have a linear topology between nodes 0 and 1. Our framework allows the intruder to listen to all the wireless channels, thus there is no need to include the intruder in the topology.

Even though the topology specified is static, our intrusion model allows mimicking dynamic

topologies. We discuss how this is possible in the description of the intruder model in details in Section 5.3.1.

5.1.5 Number of Iterations

The sensor network security protocols are usually intended to run for as long as the resource constraints of the motes can handle. In other words, most of the protocols (and sensor network applications in general) tend to put no bound on the number of executions of the protocol, leaving the protocol to run as much as the motes can survive.

While it is possible to implement such behavior, verifying such infinite system is not possible in model checking. The main reason is simply that infinite executions will lead to an infinite model, thus leading to the state explosion problem described earlier. Therefore, a bound is required to make the model finite.

We chose to make the bound as the number of firings the mote's timer is triggering. The user should specify how many timer firings represent one iteration of the protocol execution. To define the timer bound, the user has to specify how many timer firings represent one execution iteration using the special word `iteration`. In case there is no timer used in the protocol implementation, this bound is not necessary and should not be specified. For instance, in the example in the figure, there was no timer used so we did not need to add the `iteration`.

5.1.6 Objectives

An objective is a linear temporal logic formula (LTL) [44] with some additional syntax. An objective can be a literal, or a negated objective (`!`⟨o⟩) enclosed in parenthesis (`(`⟨o⟩⟨`)`⟩), or opening and closing brackets followed by an objective (⟨o⟩).

An objective may also consist of two sub-objectives combined by logical and (⟨o⟩ `&&` ⟨o⟩), logical or (⟨o⟩ '`||`' ⟨o⟩) or implication (⟨o⟩`->`⟨o⟩). These operators have the same meanings as their LTL counterparts and they are translated accordingly.

A ⟨condition⟩ allows objectives to be expressed in terms of commands and events in the protocol implementation. A ⟨condition⟩ can either describe an action done by an intruder or

a legitimate node. If the action is done by an intruder, then this is represented using the `Intruder` special word, otherwise it is a legitimate node, so the ⟨condition⟩ starts directly with the module name where the function (command/event) is implemented (⟨mname⟩). This is followed by the interface name of the function (⟨iname⟩), followed by the function name (⟨fname⟩) that may take arguments (⟨form⟩). The arguments can be either a normal argument, or in the case that the command is for sending, it can be one of the message types declared in the annotation (⟨mtype⟩).

In lines 17-20 of Figure 5.3, the objective named `Auth` is that if the intruder sends a message of type `Data`, then any legitimate node should not send a message of type `Ack` in return. The intruder is described using the `Intruder` special key word. The intruder can use the functions used by legitimate users for sending as shown. For specifying legitimate nodes activity (line 19), we directly write the module name `SensorM` where the function of sending was called, followed by the interface of sending `Send`, followed by the command `send`. Note that the parameters can be of the message types declared in the annotation language (in this example messages were declared in lines 1 and 7).

5.2 Protocol Model Generator

The protocol model generation phase replaces the original code generation phase of the nesC compiler on top of which our framework is built. It translates the protocol implementation into a Promela model.

There are many challenges in extracting a model from the implementation. In this section, we describe the most interesting challenges in the model extraction and how *Slede* deals with it.

5.2.1 State Space Explosion

As mentioned in Section 5.1.4, the major challenge in extracting the model from the implementation is to maintain the size of the model as small as possible, otherwise the verifier may never halt while verifying the model, a problem known as the *state space explosion*.

In order to generate a model with small state space, we follow two approaches. The first approach is to translate only the application level code. This way, only the code that implements the behavior of the protocol is translated. This is done by Mapping TinyOS modules to Promela constructs. For example, let the protocol to be verified contain a call to command `Send` that is required to send a message. When the nesC compiler compiles this protocol, it compiles along with it all the necessary modules required to actually send the message through the physical layer of the sensor. As our goal is to verify security of the protocol, and not how the sending modules are behaving, whenever there is a call to the `Send` command, our Promela code for sending a message is inserted in the model.

The second approach to avoid the state explosion is to put boundaries on the generated model. Our current prototype requires a bound on the number of nodes involved in the protocol, the topology of the network as well as the number of execution iterations of the protocol that can be described using our annotation language discussed in Section 5.1.

5.2.2 Discrepancy between Implementation and Modeling Languages

The modeling languages are usually less complex than implementation languages. As their goal is to model the behavior of the system to be verified, modeling languages tend to exclude many of the constructs provided by the implementation languages that are required to actually implement the system. For instance, constructing a message in nesC may require some C operations like `memcpy` or pointer arithmetic, while in modeling language there are no similar constructs.

In order to solve this problem, we make use of the embedding code feature provided by Spin version 4.0 and higher. Using the Spin's `c_code` construct, we can embed C code in the Promela model. Since nesC is a subset of C language and since nesC compiler translates the nesC code into C, our prototype overcomes the problem of discrepancy between the implementation and the modeling languages by embedding the protocol implementation at the application level in the model.

5.2.3 Translating TinyOS Specific Features

To ensure accurate verification results of the security protocol implementations, the extracted model should have the same semantics as the implementation. In addition to the fact that nesC is based on C, there are specific features related to TinyOS that need to be translated the exact way they behave. These specific features are tasks and events. We describe how we translate both of them to Promela.

5.2.3.1 Tasks

Tasks are functions whose execution is deferred till the scheduler has nothing to run. When a component *posts* a task, the task is placed in a queue. The scheduler checks the queue when it is idle to run the queued tasks. Tasks run to completion and do not preempt each other.

As mentioned earlier in Section 5.2.1, *Slede* only translates application level code in order to reduce the size of the generated model. Since the scheduler is not part of the protocol (it is part of TinyOS), we need to emulate the behavior of the scheduler in the generated model. In order to do so, we enumerate at compilation time all the tasks in the implementation. Whenever a component posts a task, the number assigned to the corresponding task at the compilation time is added to the queue of the corresponding node. At the end of the execution of the function (command/event) that posted the task, the framework executes the tasks whose corresponding numbers were added to the queue in FIFO order.

5.2.3.2 Events

TinyOS event handlers (i.e. firing Timer) are signaled from within the code of TinyOS. Unlike user-defined event handlers that are explicitly signaled in the protocol implementation which can be easily translated as a function call, TinyOS event handlers are triggered from code that is out of scope of the protocol, thus not of interest to the framework to translate. In order to deal with this problem, we create a Promela process (which is similar to a thread) that contain all the TinyOS event handlers. Every event handler is triggered when its guard expression is true. For instance, when calling `Timer.start` to start the Timer, the guard

expression for the event handler of `Timer.fired` is set to true, thus the timer is fired. Similarly, the event handler for receiving a message is triggered whenever there is a sent message in the environment. Since the event handlers are run under a different thread than the protocol, they can interrupt the normal execution of the protocol at any point of time, which is the way events handlers behave in the real protocol execution.

5.3 Intruder Model Generator

In order for the generated intruder model to behave maliciously, the intruder should be able to receive messages, read their content and send false data. To be able to do so, the intruder needs to know the types of messages exchanged between principals in the protocol and the structure of the message i.e. which fields in the message represents information about sender, receiver, data, etc so that it can read and alter the message contents.

For example, imagine a simple protocol that consists of two message types: ping and ack, each of which contain a source and destination address and a sequence number. In order to attack this naive protocol, the intruder may intercept the messages to the destination, exchange the source and destination address and reply back with this modified message as one possible attack. In order to automatically generate this attack the intruder needs to be aware of the structure of these message types in the implementation so that it can retrieve and manipulate fields in the messages that are exchanged by the protocol e.g. sender and destination address. The framework can get this information from the annotation language through which the message structure is analyzed.

5.3.1 Current Intruder Model

The intruder model that the framework currently generates follows the Dolev-Yao style [14]. An intruder in this model can read all messages and modify on their contents. However, if the message is authenticated using a MAC, then the intruder does not modify on it except if it has the MAC key; otherwise, if the intruder modifies on the content without knowing the key, the recipient will easily detect that the message was modified during its transmission. The

intruder can know if the message is authenticated using MAC through the information about message formats described using our annotation language (described in Section 5.1).

The intruder can also save messages in order to replay them. It can save only one message of each message type used in the protocol. So, for a protocol of message types ping and ack, the intruder can only save one message of type ping and one of type ack at a time. This limitation is necessary to limit the state space, otherwise, with no bounds on the number of messages to be saved by the intruder, the model will always be infinite.

The capability of the intruder to drop the packets allows mimicking the dynamic topology property of the sensor networks. Even though the users are required to specify a topology before the verification, an intruder that does not allow a message to be received by the receiver (which can happen in real life by transmitting to destination at the same time the message source is sending the message) can represent a message that runs out of power, thus modifying on the network topology. In a sense, a sensor not receiving a message or not being able to make its message be received by its destination is an isolated node (failed node). Thus, once a node in the network fails, the protocol deals with this by trying to find different paths to connect the network. This way, our intruder model with its capability of dropping messages allows verification against dynamic topologies.

Even though our framework currently supports the generation of intruder models according to Dolev-Yao model, our framework can easily be extended to provide generation of other intruder models such as node capture attack according to different patterns. As Figure 5.1 shows, we intend to allow the users of our framework to add new intruder patterns to be used for generating intruder models. This extensibility feature would help achieve more thorough verification against different types of attacks.

5.4 Verification and Counterexamples

The generated model containing the model of the protocol implementation, the intrusion model and the environment models are given as inputs to the Spin model checker [25], which verifies whether the model violates the objectives stated using our annotation language. If the

objectives are satisfied, the protocol is verified as secure. Otherwise, Spin produces a counter example that violates the security objectives. This counter example is then translated to a sequence of nesC statements. The protocol verification may not terminate if the Promela model is too large.

CHAPTER 6. Evaluation

In this chapter, we evaluate our framework prototype. We then describe verification of two sensor network security protocol implementations using our framework. For both protocols, the framework was able to find flaws in the implementation. All experiments described in this chapter were conducted on a Dell PowerEdge 1850 with dual 3.8 GHz processors and 2 GB RAM. The version of SPIN used for these experiments was 4.2.7.

6.1 Verification of the One-way Key Chain Based One-hop Broadcast Authentication Scheme

6.1.1 Protocol Overview

The one-way key chain based one-hop broadcast authentication scheme was proposed by Zhu et al. [55]. During the initialization step of this protocol, every node (denoted as A) generates a one-way key chain of certain length; that is, k_n, $k_{n-1} = h(k_n)$, $\cdots$, $k_1 = h^{n-1}(k_n)$, $k_0 = h^n(k_n)$, where $h(.)$ is a secure hash function.

The protocol then proceeds as follows: A transmits the first key of the key chain (i.e., k_0) to each neighbor separately, encrypted with the pairwise key shared between A and this neighbor. When A broadcasts its first message m_0, the message is authenticated with k_1; that is, m_0 is broadcast with message authentication code (MAC) $h(m_0, k_1)$. After the broadcast, k_1 is released alone or with the next broadcast message, which is authenticated with the next key in the key chain (i.e., k_2). To generalize, the i^{th} message m_i is broadcast along with $h(m_i, k_{i+1})$, and k_{i+1} is released after the broadcast.

6.1.2 Known Flaw in the Protocol

As pointed out by Zhu et al. [55], the adversary can launch the following attack: First, the adversary prevents a neighbor of A (denoted as B) from receiving the packet from A directly. This can be achieved by, for example, transmitting to B at the same time when A is transmitting message m_i and when A is releasing authentication key k_{i+1}. Second, after knowing k_{i+1}, the adversary sends a modified packet to B while impersonating A. Note that, the adversary has already got the released authentication key before transmitting the modified message to B, hence B will accept the modified packet. To defend against an outsider (not a neighbor of A) from launching the above attack, the original authentication scheme can be enhanced as follows: A shares a cluster key KC with all its neighbors; when A broadcasts message m_i, the MAC of the message will be $h(m_i, k_{i+1}XORKC)$. However, the defense will not be useful if the adversary has obtained KC by compromising a neighbor of A.

Verification using Slede To test if our prototype can automatically detect the above attack, we verified an implementation of this protocol with respect to a property informally stated as follows: "if a malicious node sends data, the receiver should detect that the sender is an intruder."

The verification setup including the verified property is shown in Figure 6.1 using our annotation language. Two message structures are defined, `KeyMsg` that sends the keys, and `DataMsg` that sends the message authenticated using the next key to be broadcast (the field MAC responsible for authenticating the message is declared as `private` as shown in line 11). Note that the field `info` in the implementation message structure `IntMsg` holds the key in the first message type `KeyMsg` (line 4) and also holds the data for the message of type `DataMsg` (line 9). This is one of the main reason message structure mapping is required since the fields of the message structure in the implementation can be reused in different message types.

The field `type` in the message structure has a value of 1 or 2 corresponding to messages of type `KeyMsg` and `DataMsg` respectively (lines 12-15).

To limit the size of the extracted model, we state the number of timer firings that represent

```
1  message KeyMsg mapsto IntMsg {
2  sender mapsto src;
3  receiver mapsto dest;
4  data mapsto info;
5  }
6  message DataMsg mapsto IntMsg {
7  sender mapsto src;
8  receiver mapsto dest;
9  data mapsto info;
10 private data mapsto MAC;
11 }
12 msgtypes mapsto IntMsg.type {
13 1: KeyMsg;
14 2: DataMsg;
15 }
16 iteration: 3;
17 node 0 { 1; }
18 node 1 { 0; }
19 objective Auth {
20 Intruder:SensorM.Send.send(DataMsg) ->
21    !SensorM.Leds.greenOn();
22 }
```

Figure 6.1 Verification Configuration for One-way Key Chain Based One-hop Broadcast Authentication Scheme [55]

one protocol execution (line 16). In this case, we have 3 timer firings, which is the minimum number of firings that makes one complete execution of the protocol. For the first timer firing, the node sends the key, then the message authenticated with the next key in the key chain is sent in the second timer firing, and finally the key with which the message was authenticated is sent in the third timer firing, thus making a one complete protocol execution.

The objective (lines 20-22) uses the special word `Intruder` to represent the intruder model generated by the framework as described in Section 5.1. Notice that the intruder can send messages using the command `Send.send()`, which makes describing the objective easier since it is in terms of commands/events of the protocol implementation. In our sample nesC implementation for this protocol, the legitimate nodes turn their green leds on when they receive a message from an authenticated source. Thus the objective ensures that a legitimate node should *not* turn its green led on if a message of type `DataMsg` is sent (intercepted and modified) by an intruder, otherwise the intruder would have successfully impersonated a legitimate node and fooled the legitimate node.

Our approach was able to detect this attack. The performance results for verifying this protocol are discussed in Section 6.3.

6.2 Verification of the μTesla protocol

6.2.1 Protocol Overview

μTESLA [43] was proposed for securing broadcast in sensor networks. This protocol assumes a network model that consists of a broadcast sender (e.g., base station) and multiple

```
1  message Timestamp mapsto IntMsg {
2  sender mapsto src;
3  data mapsto info;
4  }
5  message DataMsg mapsto IntMsg {
6  sender mapsto src;
7  data mapsto info;
8  data mapsto timestamp;
9  private data mapsto MAC;
10 }
11 message KeyMsg mapsto IntMsg {
12 sender mapsto src;
13 data mapsto info;
14 data mapsto timestamp;
15 }
16 msgtypes mapsto IntMsg.type {
17 1: Timestamp;
18 2: DataMsg;
19 3: KeyMsg;
20 }
21 iteration: 3;
22 node 0 { 1; }
23 node 1 { 0; }
24 objective Auth{
25 Intruder:SensorM.Send.send(DataMsg) ->
26    !SensorM.Leds.greenOn();
27 }
```

Figure 6.2 Verification Configuration for μTesla protocol [43]

receivers (e.g., ordinary sensor nodes). On receiving a broadcast message, each receiver needs to verify whether the message is really from the sender and not tampered by any intermediate nodes. The correct working of the protocol relies on the assumption that all principals (base station and ordinary sensor nodes) are loosely time synchronized. An implementer of the μTESLA protocol may not fully understand the necessity of implementing secure time synchronization for the security of the protocol, which is not explicitly specified in the description of the protocol itself. Hence, the implementer may choose to implement a simple but not secure time synchronization protocol as the foundation of the μTESLA. As elaborated in the following, our approach can automatically detect such flaws in the implementation.

6.2.2 Verification Using Slede

The verification resulted in the scenario shown in Figure 6.3. For time synchronization, node A sends out its time stamp t0, which is intercepted by some malicious node I. Node I changes the time stamp to be t0-2, and then forwards it to node B. Since the time synchronization protocol is attacked, the clock in node B will not get synchronized with node A. Later, when node A broadcasts message m1 (which is authenticated with key k1) at time t1, the message is intercepted by node I who will not further forward it. At time t1+1, when node A releases key k1, the key is also intercepted and held by node I. Right after that, node I forges a message m1' authenticated with key k1, and forwards it to B. Then, node I releases key k1 at t1+2. Upon receiving k1, node B will accept message m1' since it can be verified with k1 and the time stamp of the message (i.e., t1) is within the valid scope for acceptance.

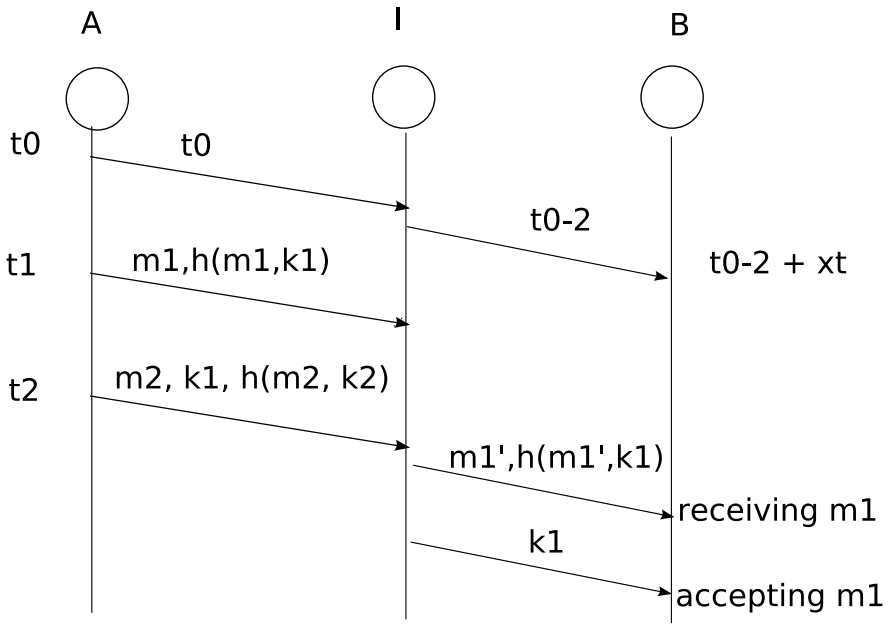

Figure 6.3 Assumption Violation in μTesla Implementation

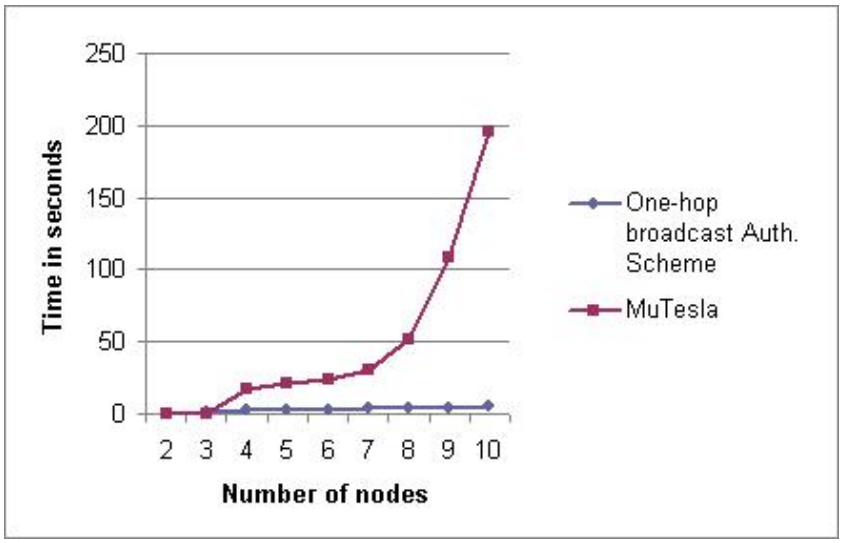

Figure 6.4 Performance in terms of time

6.3 Performance

In this section, we describe the performance of our framework in terms of number of states generated, memory used and time taken to detect the flaws. In Table 6.1, the properties of

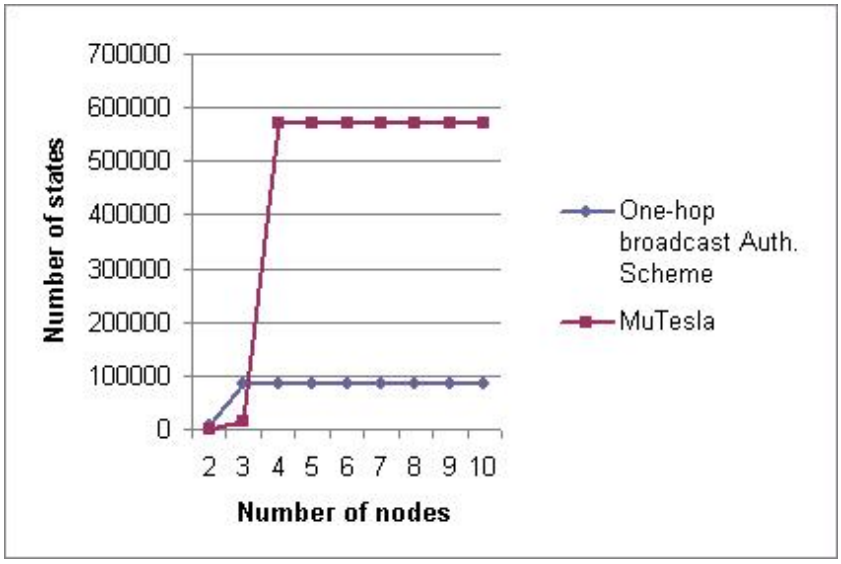

Figure 6.5 Performance in terms of states

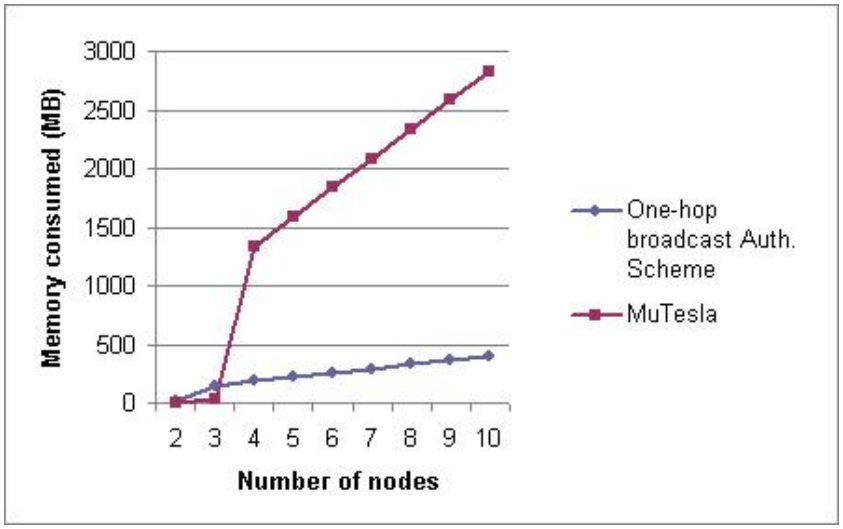

Figure 6.6 Performance in terms of memory

the sample implementations we used to verify the two protocols are displayed. The implementation of the μTesla protocol has more lines of code and more number of fields in the message declarations than the one-hop broadcast authentication scheme.

In the performance charts in Figures 6.4, 6.5 and 6.6, when we verified the one-hop authentication scheme against linear topology, we can see that once the flaw is detected, there is no significant increase in the number of generated states, consumed memory or time taken to

	Lines of Code	Message Declarations	Message Fields
One-hop Broadcast Authentication Scheme	180	2	4
μTesla	196	3	4

Table 6.1 Implementation properties of the protocols

verify even with the increase of number of nodes. The reason is that once the flaw is detected for a certain topology, the model checker does not find a problem detecting the same flaw for a larger number of nodes on the condition that the topology is the same. On the other hand, if the topology is changed and is more complex, then the generated model becomes larger, which leads to more time taken by the verifier to detect the flaw. For instance, when we changed the topology for a non-linear one and used 5 nodes to verify the one-hop authentication scheme, the model generated had 511,010 states, and the verifier consumed 1306.1 MB of memory and took 16.4 seconds to detect the flaw (not shown in the figure) as opposed to 84,368 states, 219.1 MB of memory and 2.6 seconds taken when verifying the same protocol implementation for the same number of nodes but with a linear topology.

The effect of topology on the size of the generated model is clear when we verified the μTesla protocol. Up to 4 nodes (including the base station), the topology is that the base station is connected to every node in the network. We can see that with the increase of number of nodes that are connected to the base station, the number of states and memory consumed is dramatically increasing. The reason is that having every node connected to the base station create a significant amount of interleaving executions between the sensors that are receiving every message from base station and process them at the same time. Starting with 5 nodes, we kept only 3 nodes as neighbors to the base station, that's why the number of states and memory consumed did not increase dramatically as when we were adding new neighbors to the base station.

We have concluded from our experiment that there are four main criteria that affect the performance of our framework: the lines of code of the protocol implementation, the number of message declarations, the number of fields in the messages exchanged and the topology of the network used in the verification.

CHAPTER 7. Conclusion

In this work, we presented our verification framework *Slede* for sensor network security protocol implementations. The key advantages of the framework is that it automatically extracts verifiable models from nesC implementations and allows automatic generation of protocol specific intrusion models from lightweight annotations. We described some of the challenges in verifying directly from the implementation and techniques taken to overcome them. Our framework confirmed the flaws in two sensor network specific protocols.

Our approach is sound and complete within bounds, i.e. if there is a fault scenario in the protocol, the framework will detect it and if the framework terminates for a network topology of given size declaring no faults, then there are no faults in this network or networks of smaller size using the given intruder model.

Security in sensor networks is an important problem. Our approach brings the advantages of explicit-state model checking to the sensor network applications, thereby paving the way to improve their security at a relatively small cost.

7.1 Future Work

In this section, we describe different avenues that we plan to explore in the future.

7.1.1 Verify Unbounded Networks

A typical sensor network can be constituted of hundreds to thousands of nodes, with no bound on the number of nodes that can join the network. As our underlying technology for verification is based on model checking, there must be always a limit on the number of nodes when verifying the security protocols, otherwise a large number of nodes will lead to a huge

amount of possible executions, thus leading to a state explosion as described in Section 5.2.1. Besides, even if the limit put on the number of participating nodes is large, there is no guarantee that the security goals are satisfied for networks with larger number of nodes. For instance, if the limit on the number of nodes is n, what is the guarantee that the protocol is secure for network of $n + 1$ nodes? Therefore, we need a way to ensure that the security goals are satisfied regardless of the number of nodes in the network.

In order to solve such a problem, we plan on modifying the underlying technology of our framework by applying an approach similar to regular model checking [7], where the protocol is represented as a regular language. Regular model checking is used for verifying parameterized systems, systems which do not have a bound on the number of participants. The idea behind regular model checking is to verify the regular language of the system, which ensures that the property is satisfied regardless of the number of participants of the system.

We are going to follow this approach by extracting the regular language of the protocol from its implementation. The alphabet of this language will be Linear Transition Systems (LTS) where the action of every sensor can be represented as a LTS. For instance, for a simple protocol with a node sending a message of type `Ping`, several nodes forwarding the message to the destination node, a destination node replying with a message of type `Ack`, and several nodes forwarding the reply to the source, there will be 3 LTS's: one for sending the message `Ping`, one for forwarding any type of message, and one for sending the message `Ack`. So the regular language for such a protocol will be $L = ping\ frwd^*\ ack\ frwd^*$. Malicious activity will also be presented as LTS. Every action that the intruder is capable of performing will be a separate LTS.

Our framework, taking the implementation as input, will generate malicious nodes that perform as intruders in terms of the implementation language. Therefore, having the implementation of both the legitimate nodes and the malicious nodes (generated with the help of our annotation language), our framework will generate the LTS of the intruder and the legitimate nodes, and extract the regular language of the protocol with the presence of the malicious intruder. So, the generated regular language will represent how the protocol is performing

with the presence of malicious intruder in the network. The security goals will be translated to words over this alphabet of LTS's. Verification will be simply to check if the goal (word) belongs to the regular language of the protocol. If the goal is a safety property (ensuring that something bad does not happen), then we want to ensure that the language does not have this word.

7.1.2 Verification of Sensor Network Lifetime

Sensor nodes are resource and bandwidth constrained. It may not be sufficient in this environment for a node to have an excellent security property at the cost of depleting system resources. For instance, while many security protocols use the public key (PK) cryptography for encryption, it is costly to use the traditional PK schemes in sensors due to power constraints on the sensors [17][1]. Therefore, we plan to verify not only that the security protocol is actually satisfying the security goals, but also that it does not deplete the sensors resources.

While there has been work on verifying the lifetime of the sensor network applications (i.e. [13]), they are based on simulation, thus they have the same problems of verification using simulation as described previously. We plan to extend our tool to not only verify security of protocols, but also the lifetime of the sensor network when applying those protocols.

7.1.3 Introduction of Node Compromise Model

As discussed in Section 5.3.1, the current prototype of *Slede* generates intruders according to the Dolev-Yao model. However, there are some attacks that are specific to the sensor network which are not covered by the Dolev-Yao model. The physical nature of sensors allows attackers to easily capture the node, which introduces new types of attacks. When a node is compromised, the attacker may [28]:

- replicate the capture node indefinitely (known as Replication Attack [41]),
- establish pairwise keys with any legitimate node in order to eavesdrop communication,

[1] Some works have provided PK cryptography algorithms for sensors [17, 30], but still most implementations use symmetric key for encryption

- inject false reports to the network,
- advertise inconsistent routing information, thus disrupting the whole network topology, and,
- drop silently legitimate reports.

While Dolev-Yao attack model can perform man-in-the-middle attacks (intercepting messages sent over the network channels), send/receive any message and act as a legitimate user of the network, this attack model cannot perform different attacks caused by node compromise [20]. For instance, node compromise can allow the intruder to get hold of the secret keys, thus it can act as a legitimate node and send false data (attack known as *insider attacks* [56, 53]). This cannot be modeled using Dolev-Yao model.

We plan to generate an intruder models that is able to launch such attacks introduced by node compromise. The challenge is to make the model generic so it can be used on different protocol implementations with the least human intervention possible.

BIBLIOGRAPHY

[1] M. Abadi. Security protocols and specifications. In *FoSSaCS '99: Proceedings of the Second International Conference on Foundations of Software Science and Computation Structure, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, pages 1–13, London, UK, 1999. Springer-Verlag.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Comput. Netw.*, 38(4):393–422, 2002.

[3] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state-space exploration. *Form. Methods Syst. Des.*, 18(2):97–116, 2001.

[4] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P.-C. Ham, O. Kouchnarenko, J. Mantovani, S. Mdersheim, D. von Oheimb, M. Rusinowitch, J. S. Santiago, M. Turuani, L. Vigan, and L. Vigneron. The AVISPA Tool for the automated validation of internet security protocols and applications. In *CAV'2005*, Edinburgh, Scotland, 2005. Springer-Verlag.

[5] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 139–152, Washington, DC, USA, 2006. IEEE Computer Society.

[6] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[7] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 403–418, London, UK, 2000. Springer-Verlag.

[8] D. Boyle and T. Newe. Security protocols for use with wireless sensor networks: A survey of security architectures. In *Third International Conference on Wireless and Mobile Communications (ICWMC'07)*, page 54, March 2007.

[9] J. Burns and C. J. Mitchell. A security scheme for resource sharing over a network. *Comput. Secur.*, 9(1):67–75, 1990.

[10] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[11] L. Buttyan. Formal methods in the design of cryptographic protocols (state of the art). Technical Report SSC/1999/38, Swiss Federal Institute of Technology (EPFL), nov 1999.

[12] Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, J. Mantovani, S. Modersheim, and L. Vigneron. A high-level protocol specification language for industrial security-sensitive protocols. In *SAPS*, pages 193–205, 2004.

[13] S. Coleri, M. Ergen, and T. J. Koo. Lifetime analysis of a sensor network with hybrid automata modelling. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 98–104, New York, NY, USA, 2002. ACM.

[14] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, mar 1983.

[15] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[16] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *VMCAI '04: Proceedings of the 5th Intl. Conference on Verification, Model Checking and Abstract Interpretation (vmcai 2004).*, 2004.

[17] G. Gaubatz, J.-P. Kaps, and B. Sunar. Public key cryptography in sensor networks - revisited. In *ESAS*, pages 2–18, 2004.

[18] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the 2003 conference on Programming language design and implementation*, pages 1–11, 2003.

[19] R. Gerth, R. Kuiper, W. Penczek, and D. Peled. A partial order approach to branching time logic model checking. In *ISTCS '95: Proceedings of the 3rd Israel Symposium on the Theory of Computing Systems (ISTCS'95)*, page 130, Washington, DC, USA, 1995. IEEE Computer Society.

[20] V. Gligor. On the evolution of adversary models in security protocols: from the beginning to sensor networks. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 3–3, New York, NY, USA, 2007. ACM.

[21] P. Godefroid. Using partial orders to improve automatic verification methods. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 176–185, London, UK, 1991. Springer-Verlag.

[22] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In R. Cousot, editor, *VMCAI'05*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379, Paris, France, Jan. 2005. Springer.

[23] Y. Hanna, H. Rajan, and W. Zhang. Slede: A domain-specific verification framework for sensor network security protocol implementations. In *ACM Conference on Wireless Network Security (WiSec)*, March 31 – April 2 2008.

[24] G. J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proceedings of the IFIP WG6.1 Seventh International Conference on Protocol Specification, Testing and Verification VII*, pages 339–344, Amsterdam, The Netherlands, The Netherlands, 1987. North-Holland Publishing Co.

[25] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[26] R. A. Kemmerer. Analyzing encryption protocols using formal verification techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, may 1989.

[27] N. Kothari, T. Millstein, and R. Govindan. Deriving state machines from tinyos programs using symbolic execution. In *IPSN '08: Proceedings of the 2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 271–282, Washington, DC, USA, 2008. IEEE Computer Society.

[28] S.-B. Lee and Y.-H. Choi. A resilient packet-forwarding scheme against maliciously packet-dropping nodes in sensor networks. In *SASN '06: Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, pages 59–70, New York, NY, USA, 2006. ACM.

[29] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[30] A. Liu and P. Ning. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. Technical Report TR-2007-36, North Carolina State University, November 2007.

[31] D. Liu and P. Ning. Establishing Pairwise Keys in Distributed Sensor Networks. *The 10th ACM Conference on Computer and Communications Security*, 2003.

[32] D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Comput. Secur.*, 11(1):75–89, 1992.

[33] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[34] W. Marrero, E. Clarke, and S. Jha. Model checking for security protocols, 1997.

[35] C. Meadows. Formal verification of cryptographic protocols: A survey. In *ASIACRYPT '94: Proceedings of the 4th International Conference on the Theory and Applications of Cryptology*, pages 135–150, London, UK, 1995. Springer-Verlag.

[36] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[37] C. Meadows. Open issues in formal methods for cryptographic protocol analysis. In *MMM-ACNS '01: Proceedings of the International Workshop on Information Assurance in Computer Networks*, page 21, London, UK, 2001. Springer-Verlag.

[38] J. K. Millen. CAPSL: Common authentication protocol specification language. In *NSPW '96: Proceedings of the 1996 workshop on New security paradigms*, page 132, 1996.

[39] nesC Compiler. http://sourceforge.net/projects/nescc.

[40] B. C. Neuman and S. G. Stubblebine. A note on the use of timestamps as nonces. *SIGOPS Oper. Syst. Rev.*, 27(2):10–14, 1993.

[41] B. Parno, A. Perrig, and V. Gligor. Distributed detection of node replication attacks in sensor networks. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 49–63, Washington, DC, USA, 2005. IEEE Computer Society.

[42] C. Perkins. Ad-hoc on-demand distance vector routing, 1997.

[43] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. Tygar. Spins: security protocols for sensor netowrks. In *Proceedings of ACM Mobile Computing and Networking (Mobicom'01)*, pages 189–199, 2001.

[44] A. Pnueli. The temporal logic of programs. In *The 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57, New York, 1977. IEEE.

[45] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11*, pages 267–276, New York, NY, USA, 2003. ACM Press.

[46] A. D. Rubin and P. Honeyman. Formal methods for the analysis of authentication protocols. Technical Report CITI Technical Report 93-7, CITI, 1993.

[47] G. J. Simmons. How to (selectively) broadcast a secret. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 108, 1985.

[48] P. G. Stefanos Gritzalis, Diomidis Spinellis. Security protocols over open networks and distributed systems: formal methods for their analysis, design, and verification. *Computer Communications*, 22(8):697–709, 1999.

[49] L. Tobarra, D. Cazorla, F. Cuartero, G. Daz, and E. Cambronero. Model Checking Wireless Sensor Network Security Protocols: TinySec + LEAP. In *WSAN'07*, pages 95–106, Albacete (Spain), September 2007. IFIP Main Series, Springer.

[50] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, 1998. Springer-Verlag.

[51] B. Warneke, M. Last, B. Liebowitz, and K. Pister. Smart dust: communicating with a cubic- millimeter computer, 2001.

[52] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wirless sensor network. In *Second European Workshop on Wirelss Sensor Networks (EWSN'05)*, January 2005.

[53] H. Yang, Y. Yuan, S. Lu, and W. Arbaugh. Toward resilient security in wireless sensor networks. In *in MobiHoc 05: Proceedings of the 6th ACM international symposium on Mobile*, pages 34–45. ACM Press, 2005.

[54] L. Yu, N. Wang, and X. Meng. Real-time forest fire detection with wireless sensor networks. In *International Conference on Wireless Communications, Networking and Mobile Computing*, 2005.

[55] S. Zhu, S. Setia, and S. Jajodia. LEAP: Efficient Security Mechanisms for Large-Scale Distributed Sensor Networks. *The 10th ACM Conference on Computer and Communications Security*, 2003.

[56] S. Zhu, S. Setia, S. Jajodia, and P. Ning. An interleaved hop-by-hop authentication scheme for filtering of injected false data in sensor networks. *sp*, 00:259, 2004.